Procedural interior generation for artificial intelligence training and computer graphics

E. D. Feklisov[1], M. V. Zingerenko[1], V. A. Frolov[1,2], M. A. Trofimov[1]
egor.feklisov@gmail.com | liahimzer@gmail.com
[1]Moscow State University, Moscow, Russia;
[2]Keldysh Institute of Applied Mathematics

Since the creation of computers, there has been a lingering problem of data storing and creation for various tasks. In terms of computer graphics and video games, there has been a constant need in assets. Although nowadays the issue of space is not one of the developers' prime concerns, the need in being able to automate asset creation is still relevant. The graphical fidelity, that the modern audiences and applications demand requires a lot of work on the artists' and designers' front, which costs a lot. The automatic generation of 3D scenes is of critical importance in the tasks of Artificial Intelligent (AI) robotics training, where the amount of generated data during training cannot even be viewed by a single person due to the large amount of data needed for machine learning algorithms. A completely separate, but nevertheless necessary task for an integrated solution, is furniture generation and placement, material and lighting randomisation. In this paper we propose interior generator for computer graphics and robotics learning applications. The suggested framework is able to generate and render interiors with furniture at photo-realistic quality. We combined the existing algorithms for generating plans and arranging interiors and then finally add material and lighting randomization. Our solution contains semantic database of 3D models and materials, which allows generator to get realistic scenes with randomization and per-pixel mask for training detection and segmentation algorithms.

Keywords: *procedural generation, machine learning, AI training, light-processing, tesselation, modeling*

1. Introduction

Since the creation of computers, there has been a lingering problem of data storing and creation for various tasks. In terms of computer graphics and video games, there has been a constant need in assets. Although nowadays the issue of space is not one of the developers' prime concerns, the need in being able to automate asset creation is still relevant. The graphical fidelity, that the modern audiences and applications demand requires a lot of work on the artists' and designers' front, which costs a lot. The automatic generation of 3D scenes is of critical importance in the tasks of Artificial Intelligent (AI) robotics training, where the amount of generated data during training cannot even be viewed by a single person due to the large amount of data needed for machine learning algorithms. A completely separate, but nevertheless necessary task for an integrated solution, is furniture generation and placement, material and lighting randomisation.

A number of industries use virtual reproductions of indoor scenes: interior design, architecture, gaming and virtual reality are a few. A computer model that understood the structure of such scenes well enough to generate new ones could support such industries by enabling fully or semi-automatic population of indoor environments.

Since development of neural networks algorithms, there is a big problem with creating training data sets. Computer vision and robotics researchers have begun turning to virtual environments to train data-hungry models for scene understanding and autonomous navigation. So indoor scene synthesis could also be used to automatically synthesize large-scale virtual training data for various vision and robotics tasks.

2. Related work (plan generation)

Interior layout generation can be divided into three main subtasks: plan, layout generation and 3D content generation. By "plan" we are going to implicate a general representation of a storey, that describes the types of rooms, their dimensions and possible neighboring chambers that is stored in a separate datafile. The "Layout" is the final blueprint showing all rooms being accurately placed and connected.

Plan generation can be done either by the means of machine learning [1], or by creating a general list of room placement that will be used to randomly assemble the result [2]. Depending on the type of layout constructor the rules can be either strict or be more of a loose guideline.

Layout assembly is the most complicated task of the three. The idea is to take a predetermined area (except for some cases) and separate it into subareas based on the previously generated plan. There has been developed a lot of different ways to solve this problem over the years, some of which are listed below.

Tiling

"Tiling" approach works similar to a toy constructors. It represents abstracts the whole area with small equal-sized chunks [3]. These pieces are usually placed in a grid and the process of layout generation comes down to placing the tiles in an appropriate manner. The blocks do not contain any information about rooms and are merely used for determining the overall shape.

Most grid based modern computer games [4, 5] nowadays use tiles for constructing environments to provide immense replay-ability and moderate challenge to the players. The method is also popular among independent developers since it allows to easily create level geometry and graphics on a tight budget. The advantage of tiling approach is simplicity and universality: tile primitives can be used in other methods for basic building blocks. The drawbacks are clearly visible structure of resulting model and difficulties with smaller then tile size objects.

Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

CEUR-WS.org/Vol-2763/CPT2020_paper_s6-5.pdf

Dense packing

"Dense packing" method makes of use of rooms with predetermined shapes and sizes and attempts to place them within a confined space with preset dimensions in an optimal manner [6]. It is based on a class of optimization problems with the same name. Rooms can be represented with tiles for ease of modeling. The advantage of ths method is that it based on a well-known mathematical problem and a lot of different solution have been developed for it. It also useful when particular size of rooms is required. The main disadvantage is that algorithm may require to regenerate room if placing them in the area is impossible and thus final solution may take a lot of time not even taking into account optimization problem complexity.

Growth

"Growth" algorithm is done in three phases [7]. In preparation the area is divided into small sections like tiles, each given an initial numeric value. Then the "seeds" of each section/room are subsequently planted in positions with highest values, each changing the numbers on the grid based on probabilities of other room types being adjacent. The rooms are later iteratively grown around their origin in a rectangular manner. All unused space is latter consumed by existing chambers. An advantage is straightforward implementation and possibility to work with non-rectangular shapes of target space. The main disadvantage of the algorithm is the inability to control the size of the rooms.

Inside-out

"Inside-out" approach (known as "growth" in some sources) is based on placing rooms in an optimal manner, succeeded with creating the outer wall of the house based on resulting shape [8]. The act placing rooms can be implemented in different ways. For example, the algorithm can choose the first primary chamber and place other rooms around it. This algorithms does not restrict the resulting area size and shape and adjacent rooms can be calculated and placed more accurately. However, chambers can be placed in an unoptimized manner resulting in visible gaps and the outer shape can turn out to be unrealistic.

Treemap

Other set of methods is performed by representing the floor plan as a graph and then recursively dividing the rectangle area into subsections until all rooms are placed [9]. One of the implementations requires building a treemap (hence the name) of the graph from a rectangle area. This approach works reliably on office building. Disadvantage of algorithms is that it can generate rooms with weird proportions, which however can be rectified by using a squarified treemap.

Machine learning approaches

"Machine learning" way is centered around building a Generative Adversarial Network (GAN) that generates floor based on a set of predefined layouts [1]. It consists of two networks: one generates layouts based on random noise, while the other compares the result to existing layout to determine whether it is appropriate. The most significant advantage of machine learning approach is that it unlike other methods takes social aspects into account by default. The disadvantage of such approach is that training data-sets are required which is fundamental problem.

3. Related work (furniture placement)

Early works in this field uses simple statistical relationships between objects [10]. The next step was a data-driven scene synthesis: learning priors over object occurrence and arrangement from examples. The first such method learned separate priors for occurrence and arrangement [11] but is limited to small scale scenes due to the limited availability of training data and the learning methods available at the time. Various related methods have been proposed, modeling object occurrence directed graphical models combined with Gaussian mixture arrangement patterns [12], and activity-based object relation graphs [13].

With the availability of large dataset of indoor virtual scenes such as SUNCG [14], new data-driven methods have been proposed. [15] uses a directed graphical model for object selection but relies on heuristics for object layout. [16] uses a probabilistic grammar to model scenes, but also requires data about human activity in scenes (not readily available in all datasets) as well as manual annotation of important object groups.

The most relevant papers at the moment use deep convolutional networks to learn priors over which objects should be in a scene and how they should be arranged [17] uses deep CNN that operate on top-down image representations of scene and synthesises scenes by sequential placing objects. [18] this paper utilise the same idea but reduces amount of inference steps.

Training synthetic data from virtual indoor scenes quickly becoming an essential source of learning data for computer vision and robotics systems. Several recent works have shown that indoor scene understanding models can be improved by training on large amounts of synthetically-generated images from virtual indoor scenes. At the intersection of vision and robotics, researchers working on visual navigation often rely on virtual indoor environments. Our model can complement these simulators by automatically generating new environments in which to train such intelligent visual reasoning agents.

Recently there was published a novel dataset for training and bench-marking semantic SLAM methods [19] based on SUNCG dataset rendered with ambient occlusion and photon mapping. Authors of [19] mainly focus on sampling trajectories that simulate motions of a simple home robot.

4. Suggested approach

The main difference of our work is that our system works not only with separate rooms, but is also capable of creating the layout of buildings itself and then fill rooms with necessary filling, which can be useful both in

the field of architecture and in generating a large amount of synthetic data for training. It also supports different lighting and material models, which makes the result photo-realistic.

Floor (plan) generation

The first route we have taken was a mix of "dense packing" [6] and "inside-out" [8] methods. A floor plan was generated based on input rules and room sizes in JSON format, then approximate dimensions for the first floor were calculated. Based on them the packing algorithm tried to fill the area with rooms, ending by drawing the outer wall around the layout and proceeding to the next floor. Instead of tiles it used a rectangle room of arbitrary size as a primitive.

1) Plan generation algorithm

This solution is based on some real-world knowledge and can be further developed to be more realistic. All the random distributions, used throughout the algorithm, have been assembled into a database manually.

1. The algorithm start with random number of floors and rooms of different types.
2. It checks whether there are enough rooms of each required type (for example bathrooms). Otherwise it goes to step 1.
3. Rooms are randomly distributed across floors, assigning specific dimensions to them. It also ensures that each floor has at least one bathroom. In addition, it adds a ladder in each floor, except for the last one.
4. The algorithm goes to step 3.
    - If some floors are empty
    - If the floor above is larger than the floor below
    - If some floors do not contain rooms except for bathrooms.
5. If the generation takes too long it goes back to step 1.
6. It goes through room floor-by-floor and randomly links them together.

An example of the resulting file:

```
"floor 1": {
    "bathroom 1": {
        "X": 8,
        "Y": 6
    },
    "link 1": "bedroom-living room",
    "living room 1": {
        "X": 92,
        "Y": 61
    }
```

2) Layout assembly algorithm

The implementation is based upon Blender and uses its API to generate final 3D layout and uses simple auto-generated shapes to approximate objects.

1. The algorithm goes through the plan we generated previously floor by floor.
2. It searches for linked rooms and assembles them into clusters. Each cluster gets its overall space calculated and the chambers then placed based on a simplified dense-packing algorithm.
3. Clusters, single rooms and a ladder are then packed within floor space similarly to step 2.
4. The outer wall is drawn around the structure, generated above.
5. The algorithm moves to the next floor. This time, the floor space is reduced by the space of the ladder from the previous floor and a hole is placed above it.

Thus, our implementation has further advantages: First it gain highly variable and realistic results, but it is more flexible than the Machine Learning approaches since it does not require gathering real data to get realistic layouts. Next, we can generate multi-store building with connecting ladders and finally, we support for non-rectangle room and floor shapes. However, our implementation has several restrictions:

1. Adding new rules (feeding as input) for plan generation can be rather challenging due to them being coupled together
2. The walls colliding with each other resulted in a visual glitch, that was hard to deal with.
3. In comparison to tile based methods our algorithm has difficulties with adding detailed geometry details to architectural elements: while tile based methods efficiently uses baked/precomputed geometry for windows, doors and e.t.c, our approach requires such geometry to be generated in the fly automatically for target layouts which is not trivial task generally speaking.

Furniture layout

For the first approximation of the creation of a virtual interior scene, a rather simple algorithm was selected for the layout of office furniture in the room.

1) Rotation layout algorithm. The idea is simple: traverse the edges of the office's perimeter. If the edge is shorter than the width of a desk, ignore it - a constraint relaxed in some of our other algorithms. If it is sufficiently long to place a desk, start from one end of the edge and lay down as many desks as possible along that edge. This algorithm is run three times with the only difference being the order in which the edges are traversed:

1.1) Clockwise: start from the edge left of the main door and run clockwise along the perimeter.

1.2) Counterclockwise: start from the edge right of the main door and run counter-clockwise along the perimeter.

1.3) Sort by length: sort the edges by length and process them from longest to shortest.

2) Left-right layout algorithm

The left right layout algorithm is very similar to the rotation algorithm. However, there are two key differences. First, it traverses all the sufficiently long edges to the left of the door edge first and then the edges to the right of the door; left and right are determined by taking a line perpendicular to the door edge, running through its center. Second, when laying down desks, it always works from the bottom up so that the resulting layout tends to be more symmetrical and closer to how our architects tend to lay out desks.

The left right layout algorithm is run twice. The first time we enforce that desks must be completely touching the wall and cannot hang off a short wall such as a mullion. That is, we ignore all walls that are less than desk width long (as described above). However, many offices have indentations, columns, and other edge conditions resulting in walls less than desk width length. Consequently, we run the algorithm again but this time we attempt to lay down desks on all edges, irrespective of their length, and we allow a desk to overhang an edge. After all the algorithms have been run, the code determines the highest capacity found.

3) Brute force layout algorithm

The brute force layout algorithm is roughly two orders of magnitude more computationally expensive and so is only run when the above perimeter-based algorithms do not sufficiently fill the space.

This algorithm assumes that for each edge, desks are either placed in a line facing the wall (FW) or they exist as a set of back-to-back bank of desks extending into the space.

The question is which edges should be set as back-to-back? As there are no obvious heuristics, we take a brute force approach, trying all possible combinations with one, two, or three edges designated as back-to-back and the remaining edges wall-facing. The examples of our algorithm can be found at fig. 1.

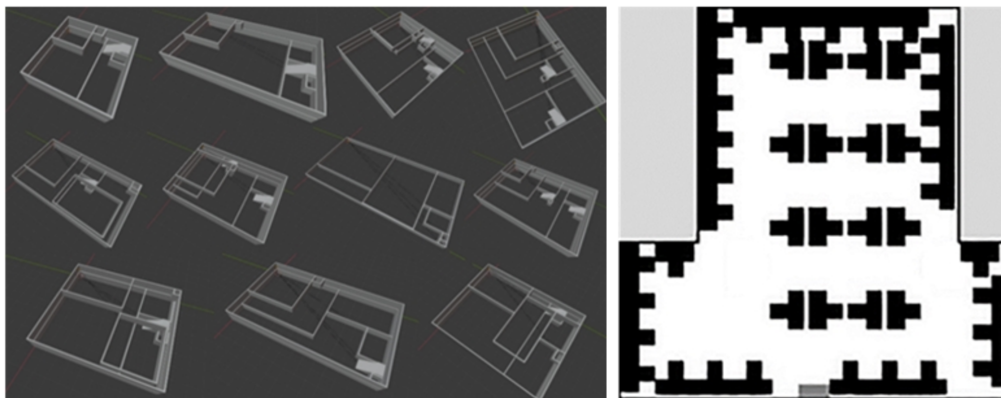

Fig. 1. Examples of generated basic 3d models of interior layout (left) and our results of our furniture placement algorithm (right)

We also try a variant where, for each edge that is longer than desk width, we consider three options: no desks, face wall, and back to back. The “no desks” option can be useful to allow a bank of desks on other walls to grow.

Unfortunately, having three options per wall leads to a combinatorial explosion in which the number of combinations to try grows very quickly with the number of walls. Thus, we only use this option if the number of walls longer than desk width is 4 or less because 34=81, which is manageable, and 35=243 which is too many for current computational resources.

Materials, lighting and rendering

This was actually one of the most time consuming problems we have to solve. The serious difficulties are concentrated around the fact that modern rendering systems use exclusively their own lighting and material models which is inconsistent with others. The realistic looking computer graphics content is created for the target rendering system and cannot be used directly in others. So, there is no such thing as open data bases of realistic 3d models due to importing/exporting 3D content from one rendering system to another is not trivial task. Taking in to account the fact of required randomization we had to build our own content creation pipeline to adopt existing 3D models. For this purpose, we used GPU accelerated open source Hydra Renderer [20]. We chose this solution because it is one of the few open rendering systems that has a full-fledged industrial level pipeline for creating content (with material conversion scripts from other popular rendering systems: VRay, Mental, Corona), while the rendering engine itself has high performance and works completely on GPU as well in Windows and Linux which is essential for training data sets generation due to large amount of required images and available Linux servers with GPUs.

For the purpose of material and lighting randomization we have adjusted the work of the artist for randomized content creation via custom 3ds max plugins that help artist to setup randomized materials and assign them to object parts (fig. 2). The artist determines the logic of randomization by setting special material parameters (fig. 2) which will later be exported to SQL-based database. This allows us to limit randomization and make it realistic in average. For example, “Target” parameter (fig. 3, down and left) acquiring some definite value allows to use this material only on a specific part of a certain class of models. We didn’t choose any modern AI based or automatic methods for 3D content generator purpose because our main requirement is high degree of control over the generated result and this is a problem for neural network based methods. Finally, we have created export tool that automatically adds all created 3d Models in our SQL-based database and then created 3D model randomizer based on this database (see fig. 4).

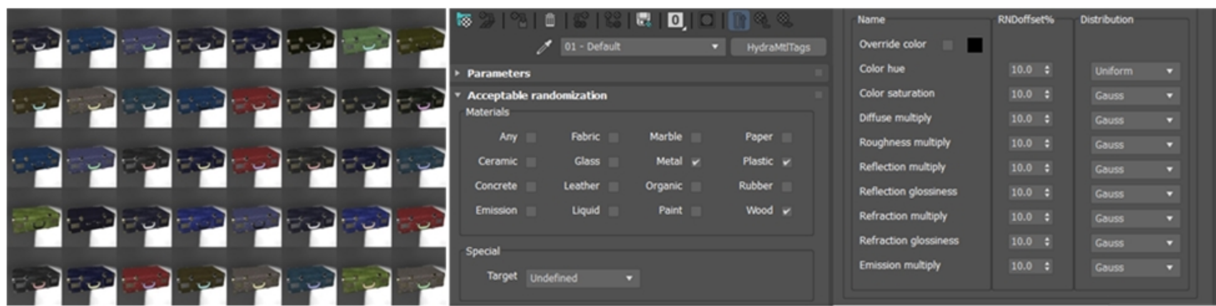

Fig. 2. Our randomization material plugin GUI and check for artist in 3ds max. This is essential for randomized results to be realistic in the target application due to artist could check whether customized distribution works in expected way or not

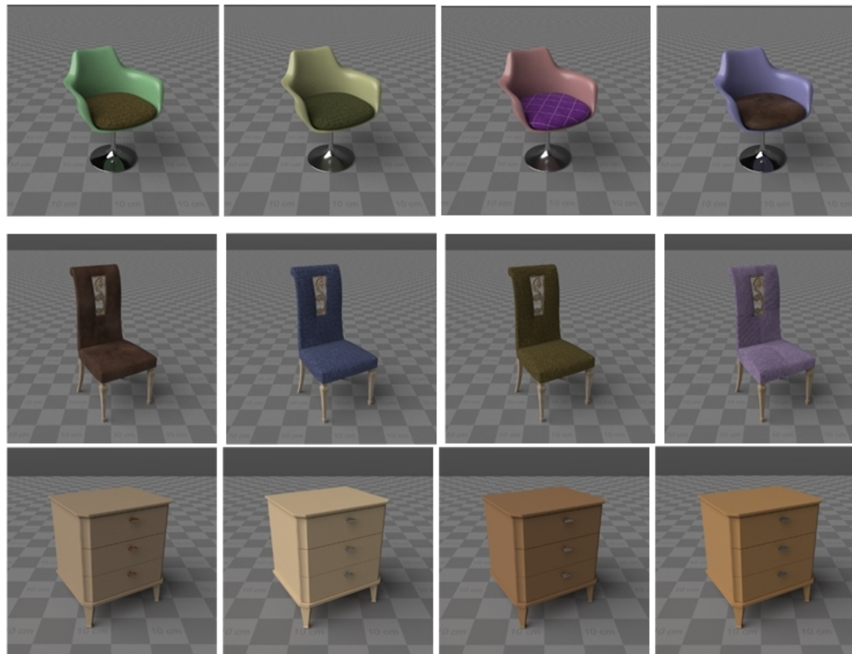

Fig. 3. Examples of randomized furniture objects from our database

Fig. 4. Early version of our furniture placement algorithm that was prototyped in Unity

Generating datasets

We used python scripts to run a specific generation scenario on the Linux server with 8 K100 GPUs. In fact, this process was not automatic because CV engineers ask very different scenarios for their experiments each time. Scripts run different parts of our generator (floor plan, furniture layout or picking 3D models from database) and connect everything together via files. Our solution is able to generate approximately 10 images per hour on a single

GPU and thus ~2 days is usually needed to generate full training dataset.

5. Conclusion and future work

In this paper we have presented procedural house interior generator that is able to produce interior images with high quality and speed. The example of generated interiors can be found at fig.5-7. However, we were not able to build complete industrial-level solution. Our system is highly fragmented connecting everything together with scripts and files, and the biggest problem is that these scripts actually have to be changed (sometimes mostly created from scratch) for each dataset generation scenario due to CV engineer's requests are very different in practice. Despite the fact that we can generate full dataset in 2 days, it takes us about 2 weeks to create new scripting scenario and debug it with the full pipeline. So we believe that using real-time rendering engines for training AI in practice is almost useless for today: the bottleneck is always in human-beings. Nevertheless, going all the way towards realistic 3D generator and rendering for AI training we would like to share our experience and state a set of problems which are, in general, not solved for today since this area of research is quite new and thus during our work we got more questions than answers.

Fig. 5. Example of render (top left), generated layout (top right), objects masks (bottom left) and object masks from in layout view (bottom right)

Fig. 6. Another examples of rendered interior layouts and object masks

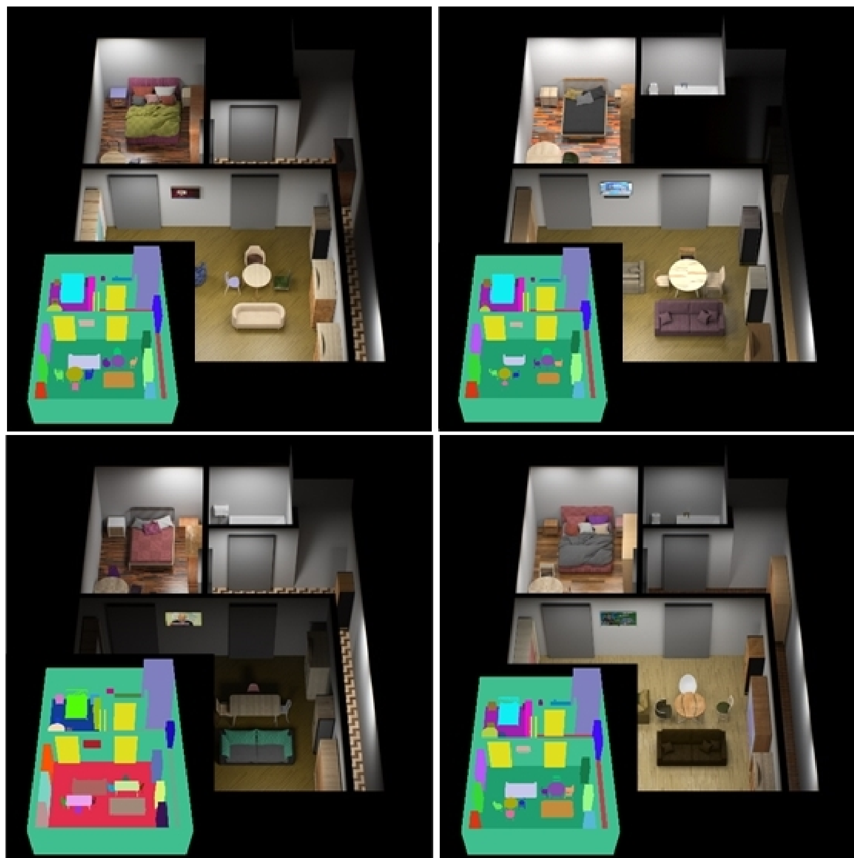

Fig. 7. Examples of different randomization result for single furniture layout and object masks

Tightly integrated framework

In our case at least 3 different people participate in dataset generation process, they are: (1) artist, who should create and check input 3D content, (2) a scripting person who creates scenarios for generator and (3) CV engineer who control the result. These people need very different skills/knowledge and we don't think that the number of participants can be reduced. However, their work could be organized better by putting them into a unit ecosystem with interface convenient for each participant. Our 3ds max plugins is the first step towards this direction, but in general this is an open problem even for a restricted area of AI training.

We used python scripts to run a specific generation scenario on the Linux server with 8 K100 GPUs. In fact, this process was not automatic because CV engineers ask very different scenarios for their experiments each time. Scripts run different parts of our generator (floor plan, furniture layout or picking 3D models from database) and connect everything together via files. Our solution is able to generate approximately 10 images per hour on a single GPU and thus ~2 days is usually needed to generate full training dataset.

Unoptimized data path, memory and disk bottleneck

In our case different algorithms (for example floor plan generation and further 3D model construction, or renderer output and further Natron post process) is communicated via files. Linux cache and fast SSD on server amortize this problem, but only a little. According to our estimates any object like mesh or image is copied from 4 to 6 times on average due to loading, storing in memory, putting to GPU or saving back to disk in different formats. This format conversion madness makes useless any attempts to speed up rendering in practice. However, we were able to optimize this process for some cases when we have formed scene library and put it to GPU once (i. e. we don't load new 3D models or images to GPU for several subsequent frames). This gives essential benefit even for our prototype with off-line rendering, but it is of critical importance for systems that's is going to use real-time rendering. We believe that generation scenario should take care of that problem in combination with some caching system and feeding the generated images directly to the neural network on the same GPU without storing it to disk (except small part of them for debug cases). We also suppose that modern denoising algorithms [21] could significantly accelerate generation process.

Absence of rendering standards and open 3D content

Available base of 3D models (like well-known ShapeNet) is not ready even for rendering: their quality is low and segmentation of parts by materials is rough. In the case of randomizing materials, we need to manually process them anyway and assign relation to our data base. Recent story with SUNCG [19] (which is far from photorealistic quality anyway) confirms the need of the open content libraries.

Procedural approaches

Unfortunately, in this work we did not manage to use procedural approaches [22] for textures, which could additionally increase the variability of the generated content.

References

[1] Merrell P., Schkufza E., Koltun V. Computer-generated residential building layouts //ACM SIGGRAPH Asia 2010 papers. – 2010. – С. 1-12.

[2] Bengtsson D., Melin J. Constrained procedural floor plan generation for game environments. – 2016.

[3] Cerny Green M., Khalifa A., Alsoughayer A., Surana D., Liapis A., Togelius J. Two-step Constructive Approaches for Dungeon Generation. – 2019.

[4] Firaxis Games Sid Meier's Civilization VI. - 2016.

[5] Triumph Studios Age of Wonders III. - 2014.

[6] Koenig R., Knecht K. Comparing two evolutionary algorithm based methods for layout generation: Dense packing versus subdivision. - 2014.

[7] Zifeng Guo, Biao Li Evolutionary approach for spatial architecture layout design enhanced by an agent-based topology finding system. - 2017.

[8] Martin J. Procedural House Generation: A method for dynamically generating floor plans. - 2016.

[9] Fernando M. Automatic Real-Time Generation of Floor Plans Based on Squarified Treemaps Algorithm. - 2010.

[10] L.-F. Yu, S.-K. Yeung, C.-K. Tang, D. Terzopoulos, T. F.Chan, and S. J. Osher. Make It Home: Automatic Optimization of Furniture Arrangement. In SIGGRAPH 2011, 2011.

[11] Matthew Fisher, Daniel Ritchie, Manolis Savva, Thomas Funkhouser, and Pat Hanrahan. 2012. Example-based Synthesis of 3D Object Arrangements. In SIGGRAPH Asia 2012.

[12] Paul Henderson and Vittorio Ferrari. 2017. A Generative Model of 3D Object Layouts in Apartments

[13] Qiang Fu, Xiaowu Chen, Xiaotian Wang, Sijia Wen, Bin Zhou, and Hongbo Fu. 2017. Adaptive Synthesis of Indoor Scenes via Activity-associated Object Relation Graphs.

[14] S. Song, F. Yu, A. Zeng, A. X. Chang, M. Savva, and T. Funkhouser. Semantic Scene Completion from a Single D. Image.

[15] V. F. Paul Henderson, Kartic Subr. Automatic Generation of Constrained Furniture Layouts.

[16] Qi, Siyuan and Zhu, Yixin and Huang, Siyuan and Jiang, Chenfanfu and Zhu, Song-Chun. Human-centric Indoor Scene Synthesis Using Stochastic Grammar

[17] Kai Wang, Manolis Savva, Angel X. Chang, and Daniel [Разрыв обтекания текста]Ritchie. Deep Convolutional Priors for Indoor Scene Synthesis. In SIGGRAPH 2018

[18] Daniel Ritchie, Kai Wang and Yu-an Lin. Fast and Flexible Indoor Scene Synthesis via Deep Convolutional Generative Models.

[19] Pavel Kirsanov, Airat Gaskarov, Filipp Konokhov, Konstantin Sofiiuk, Anna Vorontsova, Igor Slinko, Dmitry Zhukov, Sergey Bykov, Olga Barinova, Anton Konushin. DISCOMAN: Dataset of Indoor SCenes for Odometry, Mapping And Navigation. arXiv:1909.12146. September 2019.

[20] Frolov V., Sanzharov V., Galaktionov V. Open Source rendering system Hydra Renderer. https://github.com/Ray-Tracing-Systems/HydraAPI

[21] S.V. Ershov, D.D. Zhdanov, A.G. Voloboy, V.A. Galaktionov. Two denoising algorithms for bi-directional Monte Carlo ray tracing // Mathematica Montisnigri, Vol. XLIII, 2018, p. 78-100. https://lppm3.ru/files/journal/XLIII/MathMontXLIII-Ershov.pdf

[22] V.V. Sanzharov, V.F. Frolov. Level of Detail for Precomputed Procedural Textures // Programming and Computer Software, 2019, V. 45, Issue 4, pp. 187-195 DOI:10.1134/S0361768819040078

About the Authors

Egor Feklisov, student at Moscow State University, faculty of Compute Mathematics and Cybernetics, Computer Graphics and Multimedia lab. E-mail: egor.feklisov@gmail.com.

Mihail Zingerenko, student at Moscow State University, faculty of Compute Mathematics and Cybernetics, Computer Graphics and Multimedia lab. E-mail: liamhizer@gmail.com.

Vladimir Frolov, Ph. D researcher at Keldysh Institute of Applies Mathematics and Moscow State University.